Landsat 7
Mission Data and Data Pointer BCH Decoder
Prototype Description

1. OBJECTIVE AND SCOPE

1.1 PURPOSE

The BCH prototype had three purposes. The first was to understand and develop BCH error detection and correction (EDAC). The second was to determine if the minimum processing rate could be achieved using a software-only approach. The third was to determine the optimum grouping and lookup table size given the trade-off between the memory requirements and the increased speed of the larger sizes. This trade-off is discussed more thoroughly in Section 10.1.

1.2 DESCRIPTION OF PROBLEM

In the decoding process there are essentially three tasks. First, since the incoming data is interleaved within the Channel Access Data Unit (CADU), this data must be accessed in such a way that reverses the interleaving. Second, it must be determined if the codeword contains errors, and if it does, the error locations must be calculated. Third, the bits in error must be corrected and the corrected codeword must again be checked for errors. If the corrected codeword contains errors, this indicates the original codeword contained more than three errors and is therefore uncorrectable.

1.3 GOALS

The goal of the BCH decoder prototype was to process error-free data at the minimum rate of 7.5 megabits per second using C code in a UNIX environment. The prototype had to be capable of detecting and correcting up to 3 errors in 1022 bits for the mission, and up to 3 errors in 31 bits for the pointer.

2. OVERVIEW

2.1 BCH ERROR CONTROL FIELDS

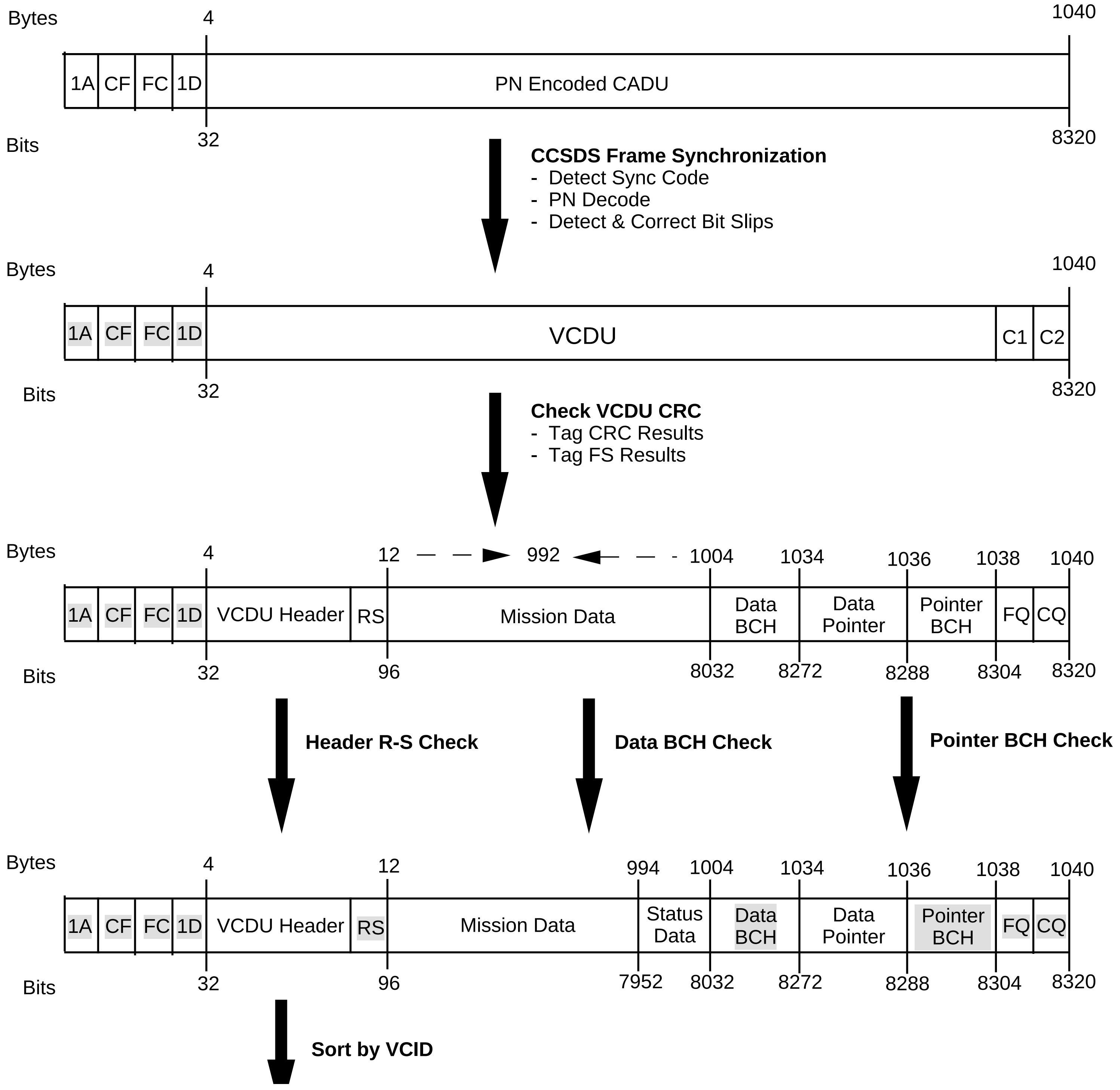
Prior to transmission from the Landsat 7 satellite to the Landsat Ground Station (LGS), 30 check bits are calculated and appended to each mission data field and 15 check bits are calculated and appended to each pointer data field. The check bits are calculated based on 992 original mission data bits, making a total mission codeword length of 1022, and 16 original pointer data bits, making a total pointer codeword length of 31. The calculations are performed by the BCH encoder, which is described briefly in Section 2.3.

2.2 BCH CODE BLOCK DESCRIPTION

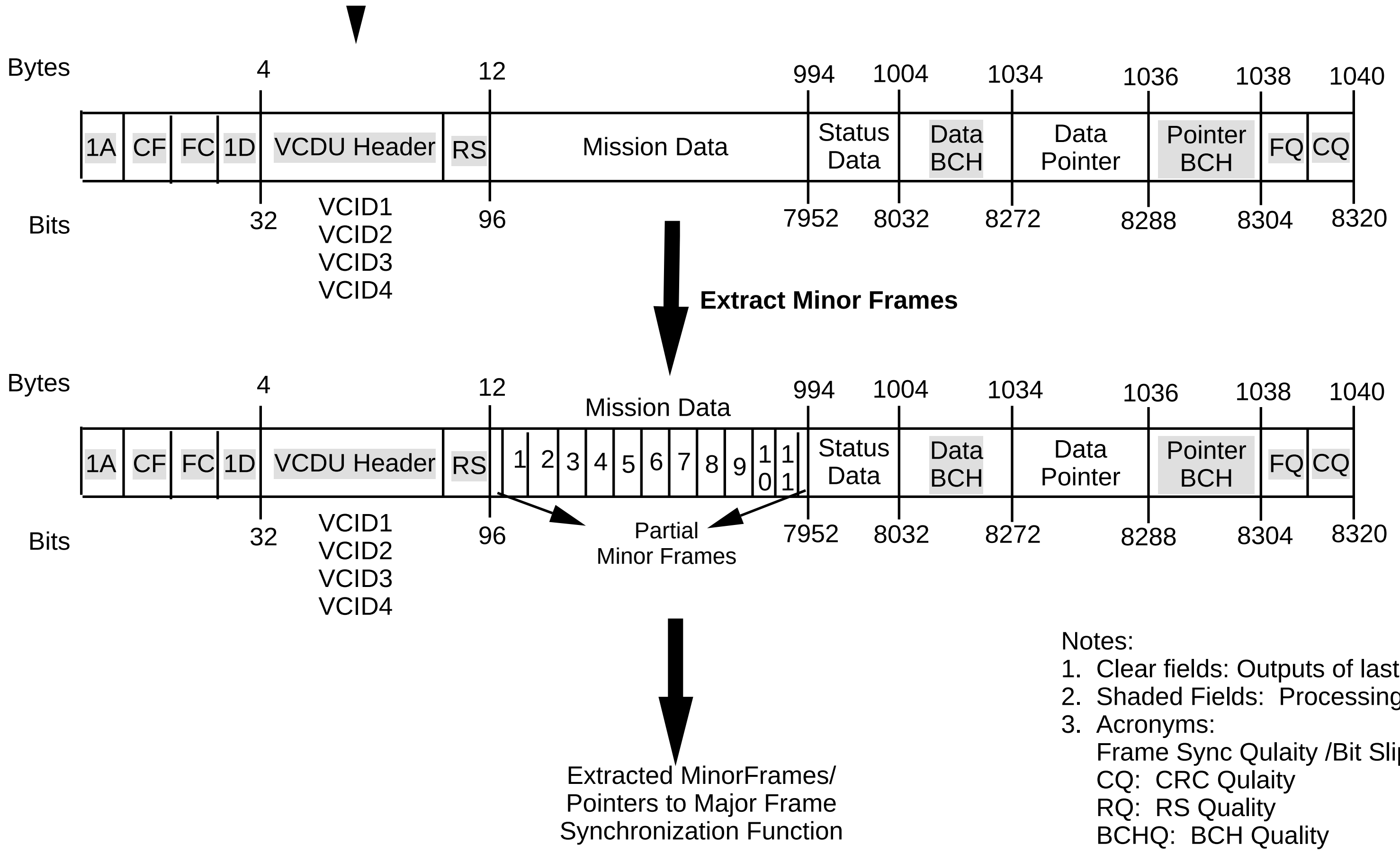
Each CADU (shown in figures 1 and 2) contains 1040 bytes of data. The mission data begins at byte 12 and continues until byte 1004 when the mission check bits begin. At byte 1034, the pointer data begins and continues until byte 1036 when the pointer check bits begin. The pointer check bits end with byte 1037. Thus the entire BCH code block contains 1026 bytes of data.

In the mission data section, from byte 12 through byte 1033, there are a total of eight interleaved mission codewords (see Figure 2). The first byte contains the most significant bit of each of the eight codewords. The second byte contains the next most significant bit of each codeword. This pattern continues throughout the mission data section ending with byte 1033 containing the least significant bit (the last check bit) of each codeword. Bit interleaving is discussed more thoroughly in Section 2.2.

The pointer codeword begins at byte 1034 and is not interleaved. The most significant bit of the pointer codeword begins immediately following the mission data section and continues for the next 30 bits ending in byte 1037 with the least significant bit and an extra bit which is ignored.



Bytes
4
1040
1A
CF
FC
1D
PN Encoded CADU
Bits
32
8320
CCSDS Frame Synchronization
- Detect Sync Code
- PN Decode
- Detect & Correct Bit Slips
Bytes
4
1040
1A
CF
FC
1D
VCDU
C1
C2
Bits
32
8320
Check VCDU CRC
- Tag CRC Results
- Tag FS Results
Bytes
4
12
992
1004
1034
1036
1038
1040
1A
CF
FC
1D
VCDU Header
RS
Mission Data
Data BCH
Data Pointer
Pointer BCH
FQ
CQ
Bits
32
96
8032
8272
8288
8304
8320
Header R-S Check
Data BCH Check
Pointer BCH Check
Bytes
4
12
994
1004
1034
1036
1038
1040
1A
CF
FC
1D
VCDU Header
RS
Mission Data
Status Data
Data BCH
Data Pointer
Pointer BCH
FQ
CQ
Bits
32
96
7952
8032
8272
8288
8304
8320
Sort by VCID



Notes:
1. Clear fields: Outputs of last process
2. Shaded Fields: Processing completed
3. Acronyms:
   Frame Sync Qulaity /Bit Slip Quality
   CQ: CRC Qulaity
   RQ: RS Quality
   BCHQ: BCH Quality

Figure 1: LPS Data Transition / Processing Stages

May 16, 1994

2.3 BCH ENCODING OF LANDSAT 7 DATA

The BCH Encoder generates the check bits for a binary message by performing polynomial division on the binary message with a generator polynomial. The check bits are the remainder of the polynomial division. The generator polynomial for the mission data takes a binary message of 992 bits and produces 30 check bits. The polynomial is:

$$g(x) = X^{30}+X^{28}+X^{23}+X^{21}+X^{19}+X^{16}+X^{12}+X^{8}+X^{4}+X+1$$

The generator polynomial for the pointer data takes a binary message of 16 bits and produces 15 check bits. The polynomial is:

$$g(x) = X^{15}+X^{11}+X^{10}+X^{9}+X^{8}+X^{7}+X^{5}+X^{3}+X^{2}+X+1$$

The BCH encoder is described in greater detail in the BCH encoder design memo finalized April, 1994.

2.4 BCH DECODING OF LANDSAT 7 DATA

The BCH decoder first creates several lookup tables which are used later for error detection. Next it reads the CADU and transposes it, which essentially reverses the interleaving. It then examines each codeword from the CADU several bits at a time (Step 1), and pulls from a lookup table values that are indexed by the bits examined. These extracted values are then exclusive-or'd and the result is three syndromes. If these syndromes are 0, there are no errors in the codeword. Otherwise, these syndromes are used to traverse a decision tree (Step 2) to arrive at the coefficients of an error locator polynomial. Next, all the values of a Galois Field (GF) table are tried as possible roots to make this error locator polynomial equal to 0 (Step 3). The index of any value that succeeds is the location of an error in the codeword. This bit is then flipped and the codeword is checked again as above for errors.

2.5 ALTERNATE APPROACHES CONSIDERED

There were two approaches considered for obtaining the syndromes in Step 1 of the BCH decoding algorithm. One goal of the prototype was to learn if the second approach would be faster than the first approach, as expected. The second approach also had an important variable built into it and the prototype was used to find the best value for that variable.

The first approach involves a GF table that has the same number of entries as bits in a codeword. Each entry is matched to a particular bit. For every bit set to 1 in the incoming codeword, the corresponding entry from the table is extracted and all extracted values are exclusive- or'd together. The final result is the three syndromes.

The second approach achieves the same result but uses a partial sums lookup table to do much of the work ahead of time. Prior to reading any incoming codewords, the extraction and exclusive-or'ing for different combinations of bits are calculated and the results stored in this table. obviously, the more bit combinations that are calculated ahead of time, the faster the processing will be once the data is actually being read because there will be less real-time exclusive-or'ing. However, the more values that are stored, the greater the memory requirements, and the greater the access time for extracting values from a very large table. The prototype was used to discover the most efficient amount of prior calculation. For more information on the use of a partial sums lookup table, see Section 7.1. For more information on the trade-off between memory and speed, see Section 10.1.

3. ASSUMPTIONS

3.1 MISSION & POINTER BCH DECODER PROTOTYPE

For the purposes of the prototype, it was understood that the incoming data would not be changed in any way. This means that bit errors were not corrected in the actual data and that the interleaved data was not deinterleaved. For the EDAC process, the incoming data did have to be accessed in such a way that it was virtually deinterleaved, but the actual data was not changed.

It was also understood that the incoming data contained mission codewords that were 1022 bits in length which includes 30 check bits. This means that the original message length was 992 bits. The BCH encoding and decoding algorithms used, however, call for an original message length of 993 bits. To overcome this inconsistency, it was agreed that the BCH decoder would add a cleared fill bit (set to 0) to the most significant bit position of the incoming mission codeword prior to EDAC processing.

3.2 INPUT TEST DATA

GTSIM provided several test data files with various combinations of bit errors. There were files containing CADU's with: clean data, data with one error in each of most significant and least significant bits for both mission & pointer, data with two errors in each of most significant and least significant bits for both mission & pointer, and data with three errors in each of most significant and least significant bits for both mission and pointer. All test data had to conform to the CCSDS format noted in Data Format Control Book.

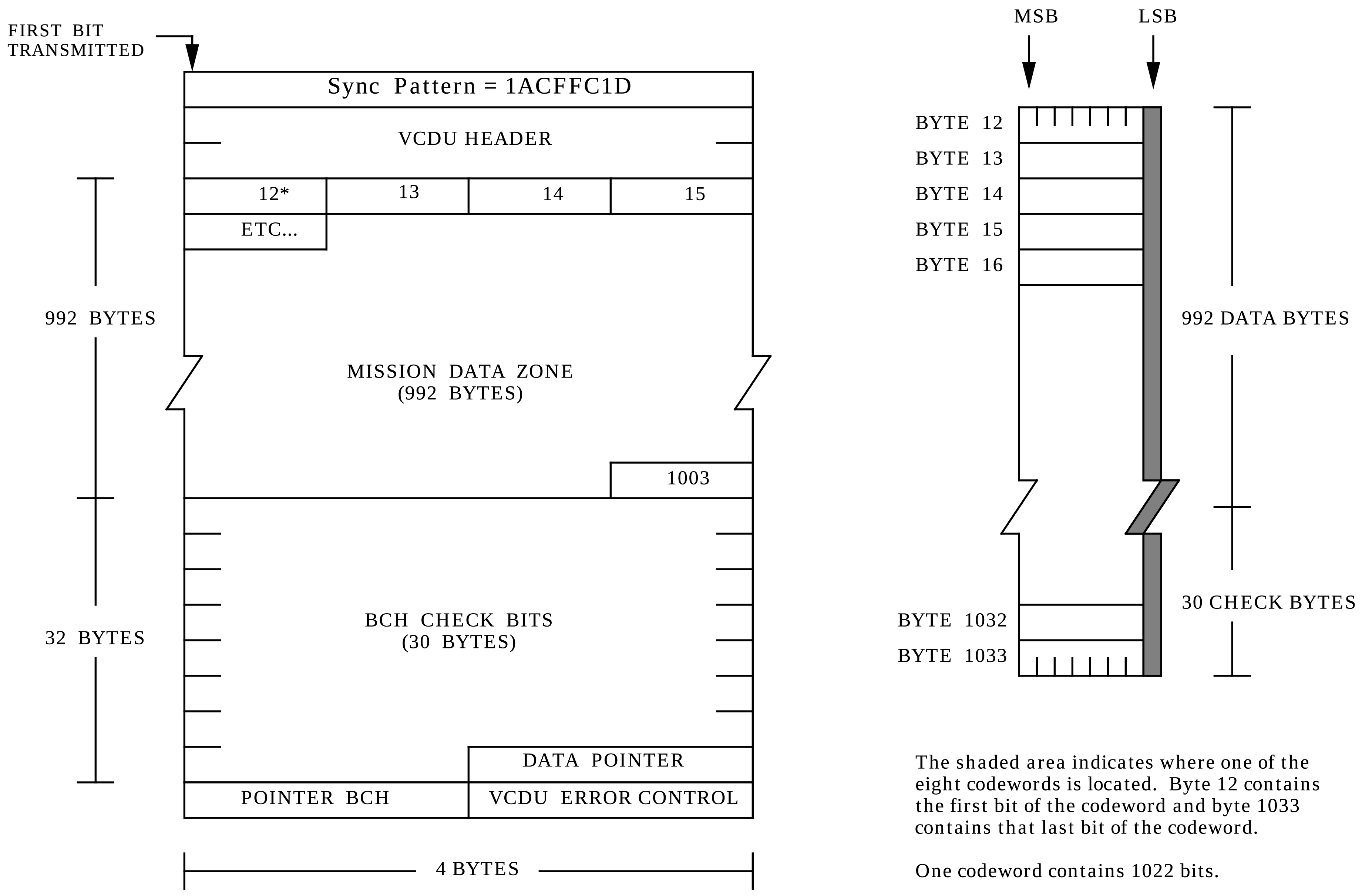
4. BCH DECODER INPUTS AND FORMATS

4.1 MISSION AND POINTER DATA

The BCH decoder prototype processes GTSIM-generated data in the form of binary files containing CADU's which contain interleaved codewords. Each CADU (shown in Figure 2) contains eight mission codewords and one pointer codeword. Specifically:

An incoming mission codeword contains 992 message bits followed by 30 check bits; 1022 bits in total. There are eight mission codewords interleaved in each CADU, resulting in a total mission code block size of 8176 bits or 1022 bytes. The block of mission codewords begins at byte 12 and ends with byte 1033.

An incoming pointer codeword contains 16 message bits followed by 15 check bits, 31 bits in total. There is one pointer codeword in each CADU and it begins immediately after the mission data block at byte 034 and ends with byte 1037.



The shaded area indicates where one of the eight codewords is located. Byte 12 contains the first bit of the codeword and byte 1033 contains that last bit of the codeword.

One codeword contains 1022 bits.

*NOTE: The numbers indicate the byte position within the CADU

The Data Pointer, Pointer BCH and VCDU Error Control are not interleaved.

Figure 2. LANDSAT 7 BCH INTERLEAVE (DEPTH OF 8)

4.2 BCH ERROR CONTROL BIT INTERLEAVING

Interleaving is a way of combining bits from several bytes that results in minimizing the overall damage of noise. When interleaving with a depth of eight, the most significant bits of eight bytes are layed out first, followed by the next most significant bits of those same eight bytes in the same order as the first. This continues until the least significant bits of all eight bytes have been layed out in order. An example of this is shown in Figure 2. The depth of eight refers to the number of codewords that are combined.

Interleaving spreads the impact of noise so that it is less likely to result in the destruction of an entire codeword. What is meant by destruction is that a codeword can no longer be corrected by the EDAC algorithm. If a CADU were arranged without interleaving so that one entire codeword followed another entire codeword, and if the EDAC algorithm, as this one, could correct a maximum of three errors in a codeword, then noise covering four or more bits would destroy a codeword. If the codewords are interleaved, however, it would take noise covering more than 24 bits to destroy one codeword.

5. BCH DECODER OVERALL DESIGN APPROACH

5.1 INITIALIZATION

In the initialization phase of Step 1, which occurs only once, several tables are created and filled. The largest tables are the partial sums lookup tables for the mission and pointer. These tables are used when calculating the syndromes in Step 1 and are explained in greater detail in Section 7.1. The other tables created aid in addition, multiplication, division, powers of 3 and 5, and inverses using GF algebra which is used heavily in Step 2. GF algebra is explained in greater detail in Appendix, Section 1. Two additional tables are created that correspond directly to the GF values. One of these tables contains the actual GF values and the other one is indexed by these values and contains the indices of the first table as its values. The second table is essentially an inverse of the first. This table significantly speeds the process of obtaining the index value of a particular entry in the GF table, something that is done frequently in Step 2. More details on table generation can be found in Section 7.1.

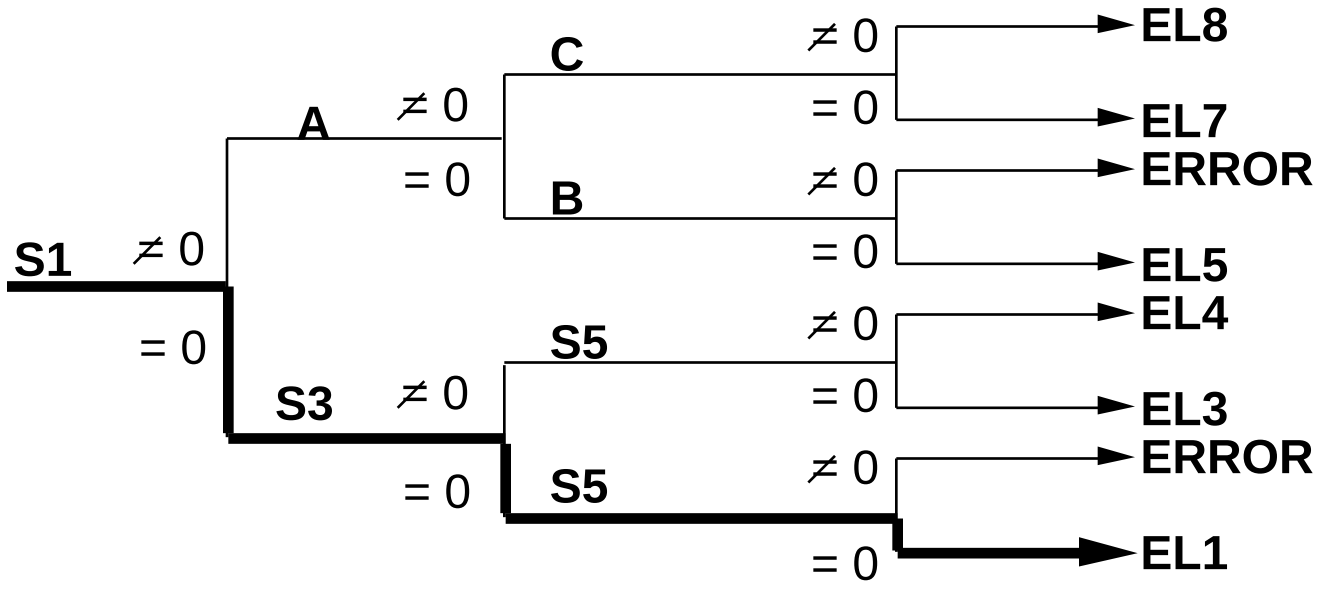
5.2 STEP 1 - COMPUTATION OF SYNDROMES

After initialization, the input data is read. Since the input data is in the form of interleaved CADU's, the first thing to do is to deinterleave the CADU. Actual deinterleaving is not necessary, but the CADU is copied and transposed in such a way that interleaving is eliminated. Next, three syndromes are calculated for each of the eight codewords in the CADU, one at a time. The syndromes are calculated in the following way: a codeword is examined by groups of bits (either 4, 8 or 16 -- set at compilation time). These groups of bits are used as indices into the partial sums lookup table and all indexed entries are extracted and exclusive-or'd together. The final result is the three syndromes, which is the output of Step 1 and the value that is passed to Step 2. If the three syndromes are all 0, however, this indicates an error-free codeword and the process skips steps 2 and 3 and begins looking at the next codeword. After each codeword in the CADU is fully processed, including the detection and correction of any errors, the next CADU is transposed and the process continues until all CADU's have been checked for errors.

5.3 STEP 2 - DERIVATION OF ERROR LOCATOR POLYNOMIAL

Step 2 uses the values of these three syndromes, and all the GF algebra tables, to traverse through a decision tree, ending at a leaf which contains the coefficients of the error locator polynomial. The decision tree is shown in Figure 3. More details of this tree-traversal are given in Section 8. The coefficients of the error locator polynomial are the output of Step 2 and are used in Step 3.

Step 2 Decision Tree



$A = S_3 + S_1^3$

$B = S_5 + S_1^5$

$C = S_5 + S_1^5 + AS_1^{-1}S_3$

$EL1 = 1$

$EL3 = 1 + S_3X^3$

$EL4 = 1 + S_5S_3^{-1}X^2 + S_3X^3$

$EL5 = 1 + S_1X$

$EL7 = 1 + S_1X + AS_1^{-1}X^2$

$EL8 = 1 + S_1X + (CA^{-1} + AS_1^{-1})X^2 + CA^{-1}S_1X^3$

Figure 3

5.4 STEP 3 - CALCULATION AND CORRECTION OF ERROR LOCATIONS

Step 3 uses the error locator polynomial to compute the error locations in the received codeword. The error locator polynomial is set equal to 0 and each of the entries in the GF table is tested as a possible root to the equation. This process, the Chien Search, is discussed more thoroughly in Section 9. The index of any entry of the GF table that satisfies the equation is the location of a bit error in the codeword. Finally, the erroneous bit(s) are flipped and Step 1 is executed again to calculate the syndromes for the corrected codeword. The codeword is error-free if all three syndromes are 0. If the syndromes are not all 0, the codeword is uncorrectable which means there were more than three errors.

6. PERFORMANCE RESULTS

One of the goals of the BCH decoder prototype was to decide on the best grouping size for both the mission and pointer data. Several tests were performed on an SGI Challenge L machine using error-free data and different combinations of grouping sizes (4, 8 or 16) for both mission and pointer data. Averages were taken and the results are shown below. The best grouping size for the mission data was definitely 8. Although there are clear advantages to using a partial sums lookup table with a large grouping size (see Section 7.1), there are disadvantages in using a very large table (see Section 10.1). The best grouping size for the pointer data was 16, although the differences here were less significant. The best combination is highlighted with an asterisk.

Grouping Size Test

Pointer Grouping	Mission Grouping	Megabits /Second
4	4	12.49
4	8	14.67
4	16	7.17
8	4	12.54
8	8	14.69
8	16	7.10
16	4	12.56
16	8	14.77*
16	16	7.16

Tests were also performed (on error-free data, with the fastest grouping combination) to determine the better optimization level between 2 and 3. The optimization level can be specified during compilation of C code on Unix-based machines such as the SGI Challenge L. Optimization level 2 produced the fastest code.

Optimization Test

Optimization Level	Megabits /Second
2	14.77*
3	13.45

Finally, using the grouping sizes of 8 for the mission and 16 for the pointer, and using optimization level 2, the following benchmark tests were performed. The data included 1,2, and 3 errors in the least and most significant bit positions. The worst case was when there were 3 errors in the most significant bit positions. This is because in Step 3, the Chien Search algorithm searches the GF table for roots to the error locator polynomial starting at the index correlating to the least significant bit position (see Section 9).

Benchmark Results

Number Errors	Megabits /Second
0	14.77
1(LSB)	10.43
2(LSB)	9.64
3(LSB)	9.51
1(MSB)	10.31
2(MSB)	.45
3(MSB)	.31

7. BCH DECODER STEP 1 DESIGN DETAIL

7.1 TABLE GENERATION

The first task in the BCH decoder prototype is the generation of all the tables -- there are 16. For both mission and pointer data, there are the GF tables and their two inverse tables, the partial sums lookup tables, and all the GF algebra tables (addition, multiplication, inverses, powers of 3 and 5, and division).

The values to be placed in the GF table for both the mission and the pointer are the elements of the GF within 2^{10} and 2^{5}, respectively. These numbers were generated separately from the BCH code and placed in two ASCII files to be read by the BCH code. More details on how these numbers are generated using primitive polynomials can be found in Appendix, Section 4. The GF inverse tables are constructed merely by reversing the entries and indices of the GF tables. For example if 11111 (binary) is located at index 15 (decimal) in the pointer GF table, then 01111 (binary representation of 15) can be found at index 31 (decimal representation of 11111) in the pointer GF inverse table. The GF table and the GF inverse table for the pointer are shown below. The GF inverse tables allow you quickly to access the index of any entry in the GF table, something that is done frequently in Step 2. The index of an entry in the GF table is also known as the "power of alpha" of that entry. The GF tables and their inverse tables are needed prior to calculating the values for the partial sums lookup tables and the GF algebra tables. More details on how the GF algebra tables are calculated are provided in Appendix, Section 5.3.

GF Table	GF Inverse Table
00001	11111
00010	00000
00100	00001
01000	10010
10000	00010
00101	00101
01010	10011
10100	01011
01101	00011
11010	11101
10001	00110
00111	11011
01110	10010
11100	01000
11101	01100
11111	10111
11011	00100
10011	01010
00011	11110

00110	10001
01100	00111
11000	10110
10101	11100
01111	11010
11110	10101
11001	11001
10111	01001
01011	10000
10110	01101
01001	01110
10010	11000
00000	01111

The partial sums lookup tables are created using the GF tables and the grouping size, which is decided at compile time. To use an example, if the pointer grouping size is 16, all permutations of 16 bits (there are 2^{16} of them) are processed as though they were the first 16 bits of the pointer codeword. In other words, for all bit positions that contain a '1', the corresponding entries in the GF table are extracted and exclusive-or'd. The result is then entered in the lookup table, indexed by the bit permutation that generated it. All permutations of the 16 bits are processed in this way. Finally, all permutations of the remaining 15 bits (of the 31 bits in the pointer codeword) are processed in the same way. If the pointer grouping size were four, there would be seven groups of the permutations of four bits, and one group of the permutations of three bits. A sample GF table and pointer lookup table for a grouping size of four is shown in Section 7.4.

The values stored in the partial sums lookup table are actually concatenations of three syndromes, S_1, S_3, S_5. For the mission data, the syndromes are each 10 bits so the three syndromes are stored in a 32-bit integer. For the pointer data, the syndromes are each 5 bits so the three syndromes are stored in a 16-bit integer. The values stored in the table are calculated differently for each syndrome. For S_1, the corresponding values are extracted directly from the GF table. For instance, if there was a '1' in bit position 30 of the pointer codeword, the entry in the GF table indexed at 30 would be extracted and exclusive-or'd. For S_3, the bit position of the '1' in the codeword is multiplied by 3 and it is this value that is used as an index into the GF table when extracting and exclusive-or'ing. For S_5, the bit position is multiplied by 5 and treated similarly. For both S_3 and S_5, if the multiplied bit position results in an index into the GF table that is out of the range of the GF table, the index is essentially 'wrapped around' until the index falls within the range of the table. Modulo 31 (for the pointer) and Modulo 1023 (for the mission) is taken of the index, which reduces it to a value within the correct range. For instance, if S_5 were being sought for a '1' in bit position 30 of the pointer codeword, 30 would be multiplied by 5 resulting in 150. Now 150 would be divided by 31, and the remainder would be used as an index into the GF table. The remainder is 26, which is within the range of the GF table. The entry indexed at 26 in the GF table would be extracted and exclusive-or'd. In this way, S_1, S_3 and S_5 are calculated for each permutation of the grouping size, and the three values are concatenated and stored in the partial sums lookup table indexed by the permutation.

Using partial sums lookup tables, much of the exclusive-or'ing necessary for deriving the syndromes is pre-calculated. In the example above with a grouping size of 16 for the pointer, there are 30 possible exclusive-or's that would be necessary if the pointer codeword were all 1's. With the partial sums lookup tables, however, 29 of these would be done during initialization, leaving only one exclusive-or to be done real-time.

7.2 DATA ACCESS

After the tables are initialized, the interleaved codewords must be accessed in the CADU. In order to do this, a structure is declared that has eight one-bit fields in it each of which can be accessed separately. This structure is then sequentially overlayed on each of the first eight bytes in the CADU, and the first bit of each of these bytes is placed in the first element of an array which will contain the eight mission codewords. The second bit from each of these bytes is placed in the second element of the mission codeword array and the third bit is placed in the third element of the array. This continues until the last bit from the first eight bytes has been place in the last element of the mission codeword array. Now the mission codeword array contains the first byte for all eight codewords. Next the bit structure is sequentially overlayed on each of the second eight bytes in the CADU and this results in the second byte of each of the eight codewords being placed in the mission codeword array. This process continues until the entire CADU has seen copied into the mission codeword array completing each of the eight mission codewords. The pointer codeword is also copied to a variable.

7.3 SYNDROME CALCULATION

Now the syndromes are calculated for each of the eight mission codewords and for the pointer codeword. This entails dividing the codewords into their various grouping sizes and extracting from the lookup tables those values that are indexed by the grouping of the codeword. All extracted values are exclusive-or'd and the result is the three syndromes contained in one 16-bit integer (for the pointer) and in one 32-bit integer (for the mission). If the three syndromes are 0, the codeword is without errors and processing stops for this codeword. If the syndromes are not all 0, the codeword contains at least one error and the syndrome values are passed to Step 2 to calculate the coefficients of the error locator polynomial.

7.4 SAMPLE SYNDROME CALCULATION

Shown below is the Pointer Lookup Table with a grouping size of four, which means the codeword is examined four bits at a time. Also shown are the indices into the table for each section of the pointer codeword. Since the grouping size is four, there are eight sections of the table. Seven sections are indexed by four bits; the last section is indexed by three bits.

index	Pointer Lookup Table	Bits Referenced
0000	0000000000000000	First four bits
0001	0101100110001110	
0010	1011010101110110	
0011	1110110011111000	
0100	0100111001110000	
0101	0001011111111110	
0110	1111101100000110	
0111	1010001010001000	
1000	1001010110101110	
1001	1100110000100000	
1010	0010000011011000	
1011	0111100101010110	
1100	1101101111011110	
1101	1000001001010000	
1110	0110111010101000	
1111	0011011100100110	

0000	0000000000000000	Second four bits
0001	0111110100101010	
0010	1111010001010110	
0011	1000100101111100	
0100	1100111100000100	
0101	1011001000101110	
0110	0011101101010010	
0111	0100011001111000	
1000	1011111011010100	
1001	1100001111111110	
1010	0100101010000010	
1011	0011011110101000	
1100	0111000111010000	
1101	0000110011111010	
1110	1000010110000110	
1111	1111100010101100	
0000	0000000000000000	Third four bits
0001	0011010111001000	
0010	0110001001101000	
0011	0101011110100000	
0100	1100000010011100	
0101	1111010101010100	
0110	1010001011110100	
0111	1001011100111100	
1000	1010110000100110	
1001	1001100111101110	
1010	1100111001001110	
1011	1111101110000110	
1100	0110110010111010	
1101	0101100101110010	
1110	0000111011010010	
1111	0011101100011010	
0000	0000000000000000	Fourth four bits
0001	1111111101111000	
0010	1101110011000110	
0011	0010001110111110	
0100	1001101100011110	
0101	0110010001100110	
0110	0100011111011000	
0111	1011100010100000	
1000	0001101111101100	

1001	1110010010010100	
1010	1100011100101010	
1011	0011100001010010	
1100	1000000011110010	
1101	0111111110001010	
1110	0101110000110100	
1111	1010001101001100	

0000	0000000000000000	Fifth four bits
0001	0011100100111100	
0010	0111000101010010	
0011	0100100001101110	
0100	1110001101010000	
0101	1101101001101100	
0110	1001001000000010	
0111	1010101100111110	
1000	1110100111011010	
1001	1101000011100110	
1010	1001100010001000	
1011	1010000110110100	
1100	0000101010001010	
1101	0011001110110110	
1110	0111101111011000	
1111	0100001011100100	

0000	0000000000000000	Sixth four bits
0001	1010011000100000	
0010	0110111110110100	
0011	1100100110010100	
0100	1101001011111010	
0101	0111010011011010	
0110	1011110101001110	
0111	0001101101101110	
1000	1000110010001100	
1001	0010101010101100	
1010	1110001100111000	
1011	0100010100011000	
1100	0101111001110110	
1101	1111100001010110	
1110	0011000111000010	
1111	1001011111100010	

0000	0000000000000000	Seventh four bits

0001	0100011010111110
0010	1000001110011000
0011	1100010100100110
0100	0010111111110010
0101	0110100101001100
0110	1010110001101010
0111	1110101011010100
1000	0101000011100100
1001	0001011001011010
1010	1101001101111100
1011	1001010111000010
1100	0111111100010110
1101	0011100110101000
1110	1111110010001110
1111	1011101000110000

000	0000000000000000	Last three bits
001	0000100001000010	
010	0001001000001010	
011	0001101001001000	
100	0010001010100010	
101	0010101011100000	
110	0011000010101000	
111	0011100011101010	

The error-free pointer codeword: 0000000010010000111000111101010 is divided into seven sections of four bits each and one section of three bits. These bits are then used as indices into the pointer lookup table (above) and the corresponding values are extracted and exclusive-or'd (below).

0000	0000000000000000
0000	0000000000000000
1001	1001100111101110
0000	0000000000000000
1110	0111101111011000
0011	1100100110010100
1101	0011100110101000
010	0001001000001010

The results are the three five-digit syndromes below (the least significant bit of the 16-bit integer is dropped). The syndromes are all 0's because the pointer codeword contained no errors.

The syndromes S_1, S_3 & S_5 are: 00000 00000 00000

If we flip the bits in positions 0, 1 and 2, the pointer codeword becomes 0000000010010000111000111101101 and now contains three errors. The values extracted from the pointer lookup table and exclusive-or'd become:

0000	0000000000000000
0000	0000000000000000
1001	1001100111101110
0000	0000000000000000
1110	0111101111011000
0011	1100100110010100
1101	0011100110101000
101	0010101011100000

The syndromes S_1, S_3 & S_5 are: 00111 00011 10101

8. BCH DECODER STEP 2 DESIGN DETAIL

8.1 TREE TRAVERSAL

The values for S_1, S_2 and S_3 derived from Step 1 are used in Step 2 to traverse the tree that is shown in Figure 3. For a zero value of S_1, the lower branch is taken. Otherwise the top branch is taken, A is calculated, then either B or C is calculated depending on the value of A. These calculations involve using GF algebra to perform addition, multiplication, inversion and exponentiation on the three syndromes. These operations are carried out using the GF algebra tables explained in Appendix, Section 5.3. $S_1{}^3$ or $S_1{}^5$ is calculated by looking up the power of alpha (see Section 7.1) of S_1 in the corresponding column (first column for exponentiation by 3, second column for exponentiation by 5) of the powers table. For addition and multiplication, the powers of alpha of the two values are looked up in the addition and multiplication tables, respectively, and for multiplication by an inverse, the powers of alpha of the two values are looked up in the division table.

Since the power of alpha of a variable is used as often as the binary value of the variable, a GF inverse table was created to speed the process of obtaining the power of alpha value. See Section 7.1 for more details on how this table is used.

After A and B or C are calculated as necessary, the tree traversal arrives at one of eight error locator polynomial designations, EL1 through EL8. Each designation represents a polynomial shown at the bottom of Figure 3. The coefficients of these polynomials provide the coefficients for the equation to be solved in Step 3. These coefficients are known as Sigma 1, Sigma 2 and Sigma 3. Sigma 1 is the coefficient of X^1 in the polynomials. Sigma 2 is the coefficient of X^2 in the polynomials, and Sigma 3 is the coefficient of X^3 in the polynomials. If the traversal arrives at position EL8, for example, Sigma 1 will be the power of alpha of S_1, Sigma 2 will be $((C * A^{-1}) + (A * S_1^{-1}))$, and Sigma 3 will be $(C * A^{-1} * S_1)$. GF algebra is used to calculate these values. Coefficients not identified by the polynomials are presumed to be 0. For instance in the case of EL5, Sigma 2 and Sigma 3 would be 0. The three Sigma values are the output of Step 2.

8.2 SAMPLE TREE TRAVERSAL

Using the values of S_1, S_3, and S_5 provided by the sample syndrome calculation in Section 7.4, the tree traversal takes the path leading to EL8 and results in the values shown below for Sigma 1, Sigma 2 and Sigma 3. Using the following values for S_1, S_3, S_5, A, B, C and both the GF Table and GF inverse table shown in Section 7.1, it is possible to see how EL8 is selected.

$S_1 = 7$
$S_3 = 3$
$S_5 = 21$
$A = 7$
$B = 11$
$C = 8$

EL8 Selected
$EL8 = 1 + S_1 * X + ((C * A^{-1}) + (A * S_1^{-1}))X^2 + C * A^{-1} * S_1 * X^3$

Next the power of alpha values shown below and the GF algebra tables (see Appendix, Section 5.3) are used to process EL8.

Power of alpha(S_1) = 11
Power of alpha(S_3) = 18
Power of alpha(S_5) = 22
Power of alpha(A) = 11
Power of alpha(B) = 27
Power of alpha(C) = 3

EL8 = $1 + 11 * X + ((3 * 11^{-1}) + (11 * 11^{-1}))X^2 + 3 * 11^{-1} * 11 * X^3$

After GF algebra is applied,
EL8 = $1 + 11 * X + 12 * X^2 + 3 * X^3$

Thus the values for Sigma 1, Sigma 2 and Sigma 3 are: 11 12 3

9. BCH DECODER STEP 3 DESIGN DETAIL

9.1 CHIEN SEARCH

The Chien Search is a relatively simple step that takes a long time to execute. This step applies the Sigma values from Step 2 to the equation below:

$$X^3 + \text{Sigma } 1 * X^2 + \text{Sigma } 2 * X + \text{Sigma } 3 = 0$$

and then tries each index in the GF table in place of X in the equation in order to find the solution. GF algebra is used to solve the equation. For three errors in the codeword, there will be three indices in the GF table that solve the equation; for two errors there will be two successful indices; there will be one successful index in the GF table for a codeword with one error.

After the error locations are discovered, the bits in those locations are flipped and the corrected codeword is checked again for errors as in Step 1. If the three syndromes are all zero this time, the correction was successful. If the three syndromes are not all zero, this means the codeword had more than three errors and is uncorrectable. In either case, processing for this codeword is now complete.

9.2 SAMPLE CHIEN SEARCH

Using the three Sigma values from the earlier sample tree traversal, 11,12,3) the equation to be solved becomes:

$$X^3 + 11 * X^2 + 12 * X + 3 = 0$$

The GF Table for the pointer is searched for indices that will make this equation true. By looking at the pointer GF Table in Section 7.1, replacing X with each index, and solving the equation using GF algebra, it is easy to see that the values at index 0, 1 and 2 do solve this equation.

Index 0:	$0^3 + 11 * 0^2 + 12 * 0 + 3 = 0$
Using GF algebra to solve:	0 + 11 + 12 + 3 = 0
Values in GF table at index 0:	00001
11:	00111
12:	01110
3:	01000

	00000

Index 1:	$1^3 + 11 * 1^2 + 12 * 1 + 3 = 0$
Using GF algebra to solve:	3 + 13 + 13 + 3 = 0
Values in GF table at index 3:	01000
13:	11100
13:	11100
3:	01000

	00000

Index 2:	$2^3 + 11 * 2^2 + 12 * 2 + 3 = 0$
Using GF algebra to solve:	6 + 15 + 14 + 3 = 0
Values in GF table at index 6:	01010
15:	11111
14:	11101
3:	01000

	00000

It turns out that these are the only indices that solve this equation. Thus the errors in the pointer codeword must be at positions 0, 1 and 2. But we know they are because these are the bits we flipped. If these bit positions are flipped back, we end up with the codeword with which we started: 0000000010010000111000111101010 and the same extracted and exclusive-or'd values from the pointer lookup table:

0000	000000000000000
0000	000000000000000
1001	100110011101110
0000	000000000000000
1110	011110111011000
0011	110010011001010 0
1101	001110011010100 0
010	000100100000101 0

The syndromes S_1, S_3 & S_5 are: 00000 00000 00000

Since the three syndromes are zero when the corrected codeword is checked for errors, we know the correct bits were flipped and we now have an error-free codeword. Thus processing for this codeword is complete.

10. ANALYSIS OF BCH DECODER PROTOTYPE RESULTS

10.1 STEP 1

10.1.1 EFFECT OF GROUPING SIZE ON PERFORMANCE RESULTS

It was explained in Section 7.1 that using the partial sums lookup tables results in fewer real-time exclusive-or's. Obviously, if the grouping size is larger, more of the exclusive-or'ing is done during initialization and less during real-time. It might be assumed, therefore, that the larger the grouping size, the faster the processing time. However, since a larger grouping size results in a larger table size, it turns out that there is a point of diminishing returns with regard to grouping and table size. Once a table size is larger than the machine cache size, page faults will occur when values are accessed from all over the table. If the table is large enough, the time consumed by these page faults will be greater than the time saved by doing most exclusive-or'ing during initialization. The optimum grouping size for the mission data turned out to be 8 and for the pointer data, the optimum grouping size was 16. See Section 6 for performance results.

10.1.2 POTENTIAL IMPROVEMENT

The overlaying structure methodology for transposing the CADU's is explained in Section 7.2. I believe the code in Step 1 could be improved by avoiding copying the CADU and simply using the overlaying structure method to access the data from the CADU directly. It would mean changing the current design slightly because each mission codeword is processed completely before the next mission codeword. If the CADU were accessed directly without copying, it would probably mean that one byte from each mission codeword would be processed before the next byte from each mission codeword. It would also mean that the code would probably be limited to a grouping size of 8 for the mission, as compared to now when it has the flexibility to range from 4 to 16.

10.2 STEP 2

10.2.1 TREE TRAVERSAL ACCELERATED BY GF INVERSE TABLE

One improvement was made to Step 2 midway through the prototyping effort. In the calculations required for the tree traversal, it is often necessary to obtain the index of a particular value in the GF table. When Step 2 was first implemented, this index was obtained each time by exhaustively searching the GF table. An improvement was made, however, in the creation of a GF inverse table (see Section 7.1). In this way, the index could be obtained immediately merely by using the original value as an index into the GF inverse table.

10.2.2 POTENTIAL IMPROVEMENT

No future improvements to Step 2 are foreseen at this time.

10.3 STEP 3

10.3.1 IMPACT OF CORRECTING ACTUAL DATA

For the purposes of this prototype, the BCH decoder does not change the actual input data. In implementation, however, the actual data will have to be corrected if there is an error and this will result in slower speeds.

It should be noted, however, that most CADU's will have no errors. Those that do will most likely not be correctable. For a codeword not to be correctable, there must be noise that covers more than 24 contiguous bits. Most of the time there will be either no noise or noise that covers more than 24 contiguous bits; both instances in which we do not correct. Thus most of the time we will not be correcting actual data.

10.3.2 POTENTIAL IMPROVEMENT

It is also possible some improvement in processing time can be obtained in Step 3 by doing some of the calculations required for the Chien Search ahead of time. As each of the entries from the GF table are tried on the error location polynomial to make it equal to 0 (see Section 9), some calculations are required. It is possible some of these calculations could be done ahead of time (at the expense of memory, of course) so there would be fewer real-time calculations required. This has not been thoroughly explored, but the possibility exists.

APPENDIX

1 - Galois Field Algebra

Galois Field (GF) algebra is similar to high school algebra or arithmetic except that GF operates within a finite field. For example, using base ten integer arithmetic, take the element denoted 7, add the element denoted 8, and obtain element 15. An integer added, subtracted, or multiplied to another integer, always results with some element in the infinite set. However, in GF algebra it is possible to take the element 7, add the element 8, and obtain the resulting element only within a finite number of elements. To learn about Galois Field algebra, we must first learn the algebraic laws governing the Galois (or finite) field. These rules are the standard algebraic laws. These laws may, however, become so familiar that some may even have been forgotten.

A field is a set of elements in which addition, subtraction, multiplication, and division can be done without leaving the set.

2 - Binary Fields GF(2)

At this point, we should have learned enough about fields and reviewed enough of the basic algebraic laws to develop a finite field. To demonstrate the idea of finite fields, we start off presenting the simplest case: modulo-2 arithmetic.

2.1 - Modulo-2 Addition, and Subtraction

Consider the set of two integers, F(0,1). Define a binary operation, denoted as addition "+", on Galois as follows:

Modulo-2 addition

Addition	0	1
0	0	1
1	1	0

This can be implemented with a single Exclusive-Or gate. This binary operation is called modulo-2 addition.

2.2 - Modulo-2 Multiplication, and Division

Because we have modulo-2 addition "+" defined over a binary group, let us develop a binary field. Define modulo-2 multiplication. Consider the same set of two integers, F(0,1). Define another binary operation, denoted as multiplication ".", on F(0,1) as follows:

Modulo-2 Multiplication

Multiplication	0	1
0	0	0
1	0	1

This can be implemented with a single AND gate. This binary operation is called modulo-2 multiplication. This set F(0,1) is a field under modulo-2 addition and multiplication. Because we now know the underlying algebraic structures to perform GF(2) arithmetic, let us talk about extension fields. We are interested in prime finite fields called Galois Fields GF(p). The previous binary operation had the minimum number of possible elements that comprised GF(2). Extension fields are GF(p^m), where m can take the values 3, 4, 5,..... Therefore we will mainly speak of binary Galois Fields GF(2) and the extended binary Galois Fields GF(2^m).

3 - Primitive Polynomials

Polynomials over the binary field are any polynomials with binary coefficients. They are binary polynomials. Each of these polynomials, denoted as f(x), are simply the product of its irreducible factors. We can create an extension field by creating a primitive polynomial p(x). A primitive polynomial p(x) is defined to be an irreducible binary polynomial of degree m that divides x^n+1 for n = p^m-1, which is equal to 2^m-1 and which does not divide x^i+1 for i is less than n. Once a primitive polynomial p(x) is found, the elements of the Galois Field can be generated. Any primitive polynomial p(x) can construct the p^m = 2^m unique elements, including a 0 (zero or null) element and a 1 (one or unity) element. A degree m polynomial f(x) over GF(2) is defined to be irreducible over GF(2) if f(x) is not divisible by any polynomial over GF(2) of degree greater than zero but less than m.

4 - Field Symbols (Alpha) ^{i}

Any irreducible polynomial that divides x^n+1 is a primitive polynomial p(x), for example, for data pointer $p(x) = x^5 + x^2 + 1$ of degree m = 5. We can now generate our Galois Field $GF(p^m) = GF(2^5) = GF(32)$. Because we use the extension of GF(p) = GF(2), the null and unity elements in $GF(2^m)$ are the same as the null and unity elements in GF(2). Now we set alpha (x) = alpha = x to obtain the 5-tuple $j4x^4 + j3x^3 + j2x^2 + j1x + j0$ for each element $alpha^i$ of $GF(2^5)$. We have

alpha $= x^1$
$alpha^2 = x^2$
$alpha^3 = x^3$
$alpha^4 = x^4$
$alpha^5 = x^5$

What do we do now to change x^5 into appropriate 5-tuple. We simply take the modulo function of the result. For example $alpha^5$ = alpha times $alpha^4$ modulo p(x). One of the ways to perform this modulo function is to set our fifth degree p(x) to zero and obtain the 5-tuple equivalent to x^5. Working this out we obtain: $x^5 = x^2 + 1$. Therefore, alpha to the fifth is equivalent to alpha to the two plus one. Alpha to the sixth is equivalent to alpha to the third plus one. In the same manner, all the 32 elements power of alpha will be computed.

5 - Addition, and Subtraction, Multiplication, and Division within $GF(2^m)$

Addition in the extended Galois Field $GF(2^m)$ can be performed by one of two methods. The most common method is by exclusive-oring the elements' vector representations position by position. This is simply performing modulo-2 addition. We are not using carry arithmetic. The least common method is by adding their polynomial representations together.

5.1 - Addition, and Subtraction Within $GF(2^5)$ and $GF(2^{10})$

For example, addition of two elements within $GF(2^5)$ is $alpha^2 + alpha^8$ = 00100 xor 01101 = 01001 = $alpha^{29}$. Since subtraction is identical to addition for example for the case of $alpha^4$ and $alpha^8$ we have:

$\text{alpha}^4 + \text{alpha}^8 = \text{alpha}^4 - \text{alpha}^8$
$= -\text{alpha}^4 + \text{alpha}^8$
$= -\text{alpha}^4 + \text{alpha}^8$

An addition table has been developed within GF(2^5) and GF(2^{10}) for (31, 16, 3) and (1023, 993, 3) BCH codes. Since $\text{alpha}^i + \text{alpha}^j = \text{alpha}^j + \text{alpha}^i$, only half of the tables need to be filled.

5.2 - Multiplication and division within GF(2^m)

As in the case of addition and subtraction in GF(2^m), we also have two methods to perform multiplication and division. The most common method is by summing the symbols' exponents modulo 2^m-1 (or modulo n) and the least common method is again the polynomial method. Using the exponent mod n multiplication method we have:

$\text{alpha}^5\ \text{alpha}^2 = \text{alpha}^{5+2}$
$= \text{alpha}^7$

5.2.1 - Examples Using Mod 15 Within GF(2^4)

For multiplication we have:

$\text{alpha}^5\ \text{alpha}^{14} = \text{alpha}^{5+14}$
$= \text{alpha}^{19}$
$= \text{alpha}^{19}$ mod 15
$= \text{alpha}^4$

Another method of performing the modulo function for multiplication or division is to keep multiplying or dividing by alpha^{15}, which is unity, until we obtain a symbol within the finite field.

$\text{alpha}^5\ \text{alpha}^{14} = \text{alpha}^{5+14}$
$= \text{alpha}^{19}$
$= \text{alpha}^{19}$mod 15
$= \text{alpha}^{19} / \text{alpha}^{15}$ for $\text{alpha}^{15} = \text{alpha}^{-15} = \text{alpha}^o=1$
$= \text{alpha}^4$

Using the exponent mod n division method:

alpha^5 / alpha^2 = alpha^5 alpha^{-2}
= $\text{alpha}^{5+(-2)}$
= alpha^3

Using the exponent mod n division method for in inverse symbol alpha^{-i}:

alpha^5 / alpha^{14} = $\text{alpha}^5\text{alpha}^{-14}$
= $\text{alpha}^{5+(-14)}$
= alpha^{-9}
= alpha^{-9} mod 15
= alpha^6

Multiplication is easily performed by adding the exponents modulo n and noting alpha^i $\text{alpha}^{-\text{infinity}}$ = (alpha^i) (0) = 0.

5.3 - Addition, Multiplication, Powers, and Inverse Within GF(2^5) for Data Pointer and GF(2^{10}) for Mission Data

5.3.1 - Addition Within GF(2^5) and GF(2^{10})

Addition within GF(2^5), and GF(2^{10}) can be performed by exclusive - oring the elements' binary representations position by position. This is done by performing modulo - 2 addition. For example, using the two elements alpha^{27}, and alpha^{19} of GF(2^5) We have:

alpha^{27} = 01011
alpha^{19} = 00110
----
alpha^{27} xor alpha^{19} = 01101 = alpha^8

Where the field was generated from the following minimum polynomial

Alpha^5 + alpha^2 + 1.

Quantities belonging to the field GF(2^m) are usually stored and manipulated as m bit computer words. The 2^m elements of any such Glois Field can be written as powers of alpha, where alpha is a primitive element of the field.

Addition tables within GF(2^5), and GF(2^{10}) for data pointer and mission data, respectively, will be generated and used in the computation. An addition table within GF(2^5) is a precomputed table of all the elements within the field of GF(2^5). The entries in the table are precomputed, and are illustrated below:

[Addition table printed from the output of the program goes here]

5.3.2 - Multiplication within GF(2^5) and GF(2^{10})

As in the case of addition in GF(2^m), m=5 and m=10 also have a method to perform multiplication. The most common method is by summing the elements' exponents modulo 2^m-1 (or modulo n), where n is the length of the code word. The n and m are related through n = 2^m-1. For m=5 and m=10, the value of n is 31 and 1023. The following example uses the exponent mod n multiplication method. Using the two elements alpha^{27} and alpha^{19} of GF(2^5) we have:

alpha^{27} * alpha^{19} = alpha^{27+19} = alpha^{46}

alpha$^{46 \text{ mod } 31}$ = alpha^{46} / alpha^{31} = alpha^{15}

alpha^{31} = alpha^{-31} = 1 / alpha^{31} = alpha^{0} = 1

The method of performing the modulo function for multiplication (or division) is to keep multiplying (or dividing) by alpha^{31}, which is unity, until we obtain an element within the finite field GF(2^5) for pointer and GF(2^{10}) for mission data.

The field of GF(2^5) has 31 elements, including zero. Zero has an alpha representation of a power equal to minus infinity. One has an alpha representation of a power equal to zero. The powers of alpha will roll over at 31 to value one.

Multiplication table within GF(2^5) is a precomputed table of all the elements within the field of GF(2^5). The entries in the table are precomputed and they are illustrated below:

[Multiplication table goes here]

5.3.3 - Powers within GF(2^5) and GF(2^{10})

This computation is a special case of the multiplication within the fields. The equation for computing the Elementary Symmetric Functions uses powers of syndromes. Using the exponent mod n multiplication, powers of all the elements within the fields can be computed. Using the element alpha^{27} within GF(2^5) we have:

alpha^{27} * alpha^{27} = (alpha^{27})^2 = alpha^{54}

alpha$^{54 \bmod 31}$ = alpha^{54} / alpha^{31} = alpha^{54} * alpha^{-31} = alpha^{23}

5.3.4 - Inverse Within GF(2^5) and GF(2^{10})

The inverse of an element within the field is computed as follows:
For example, the inverse of alpha^{27} within GF(2^5) is:

1 / alpha^{27} = alpha^{31} / alpha^{27} = alpha^{31} * alpha^{-27} = alpha$^{31+(-27)}$ = alpha^4. alpha^4 is an element within the field.

Division within the fields uses inverse and multiplication within the fields.